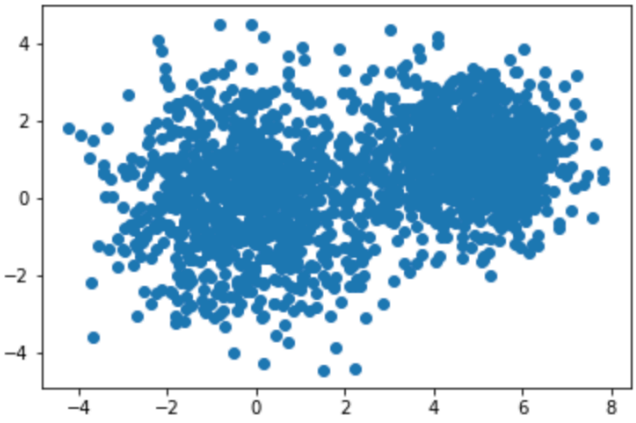
IE 529

Stats of Big Data & Clustering

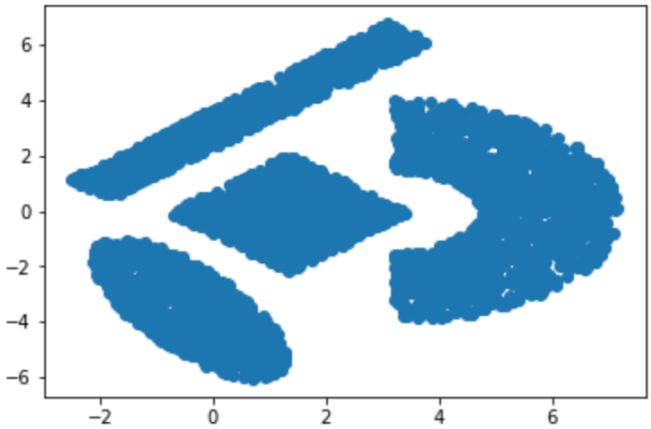
Computations: 2

1. Include a scatter plot for each set of data, as is, i.e., with no clusters identified yet.

Dataset 1



Dataset 2



2. Submit your pseudo-codes for the Lloyds/K-means algorithm, the GreedyKcenters and Single Swap algorithms, and the Spectral Clustering algorithm, each on a separate sheet of paper. State clearly whether you are considering normalized or un-normalized Spectral Clustering.

(1) Pseudo-code for Lloyds algorithm

- Initialize K centroids for the dataset given user-based K.
- Compute the distance (L2-norm) between sample and centroids, assign the centroids index to sample with the least distance between them.
- While converge condition:
  - Re-compute the centroids by compute the centroids of grouping samples.
  - Re-assignment the index of centroids of samples
- End while

(2) Pseudo-code for GreedyKCenters

- Initialize first centroids for the sample data, randomly
- While # centroids < K:
  - Let X_l belongs to the subset of sample data removing the set of centroids which maximize the distance of (X_l, C)
  - Set the new set of centroids be the set union with X_l
- End while

(3) Pseudo-code for Single Swap

- Initialize set C for K centroids
- While exist C not empty and X_i belongs to the subset of X removing C:
  - Make dist(X \ C) < dist(C)
  - Set $C = C \setminus m_j \cup X_i$
- End while

(4) Pseudo-code for Spectral Clustering

Un-normalized

- Initialize the similarity/weighted matrix W
- Initialize degree matrix D
- Initialize Laplacian matrix L = D – W
- Compute eigen/ spectral decomposition of L: U is the matrix eigenvectors, Λ is the diagonal matrix of eigenvalues
- Consider the first K eigenvectors of U (Y) -> associated with K smallest eigenvalues
- Perform K-Means on Y

3. For your K-means algorithm by itself, and for your Spectral Clustering algorithm, present the output (clustering) results for BOTH sets of test data given to you (see below).

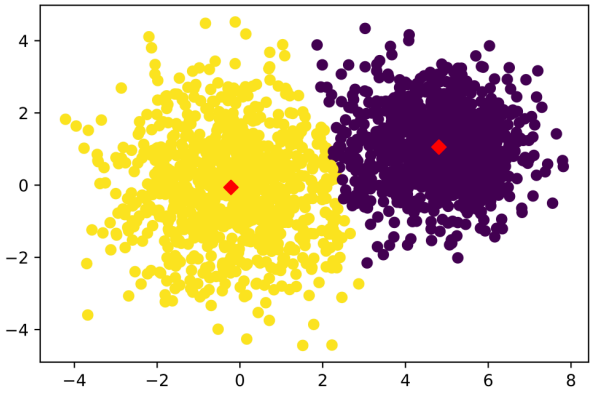
4. Evaluate the results for a range of K values (number of clusters) from K = 2, to K = 11 (but not all of these; try 3 different values, then try to pick 2-3 more that will help you find the best cluster outcome). When implementing the K-means algorithm (by itself and from your Spectral Clustering algorithm), you should try multiple initializations (say 5 to 10) and then select the 'best' result, for each K value. State how many different initializations you tried and why.

(1) The output should consist of a scatter plot of the data with the clusters differentiated by color, and with the centroids highlighted in a different color, size and shape than the corresponding cluster members.
(2) A plot of the distance metric, D, versus, K, for the best outcomes found for each of the K values you considered, for both K-means and Spectral approaches.
(3) Using the two best results from your K-means, that is, two different K-values that seem to give the best clustering result, use your GreedyKcenters algorithm to find an initialization and re-run K-means with this initialization. Describe how this does or does not improve your results.

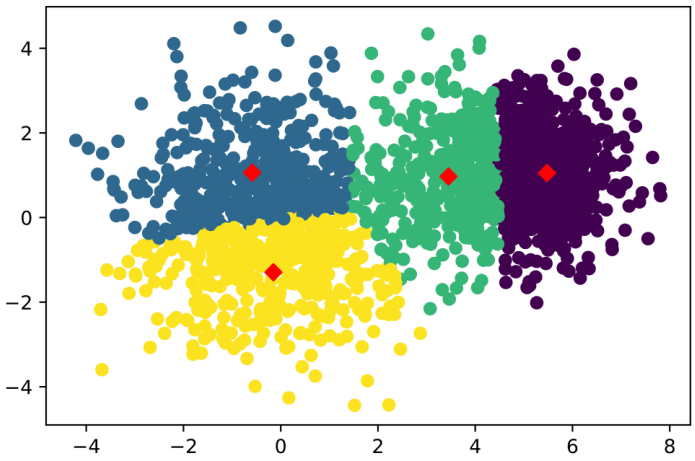
(1)
random initializations I implemented for KMeans are 10. Because the first series of centroids are randomly, arbitrarily selected. It can be anywhere even an outlier. So the more initializations, the more chance you get a correct clustering result.

KMeans only
Dataset 1

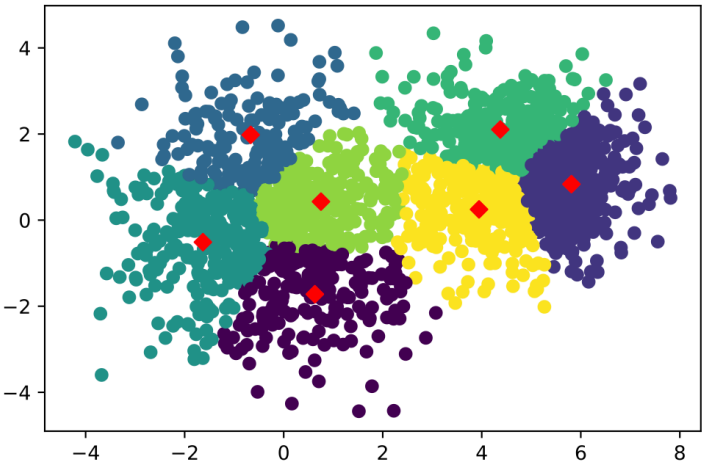
K = 2



K = 4



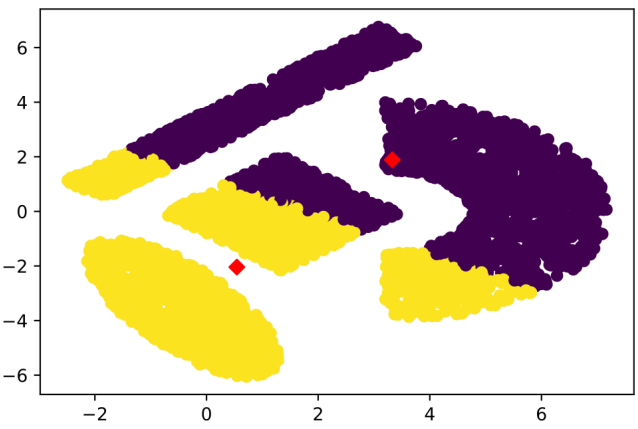
K = 7



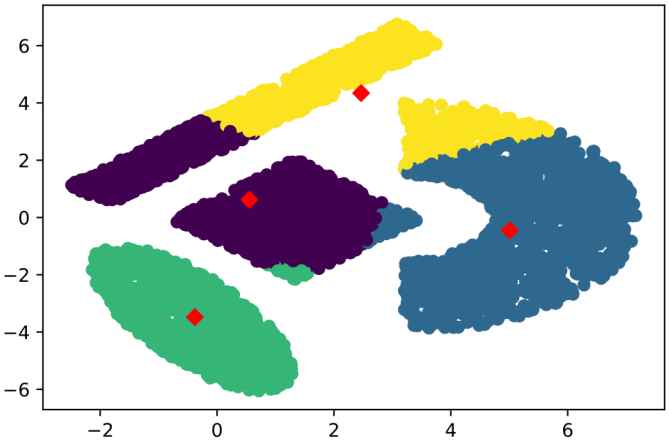
KMeans

Dataset 2

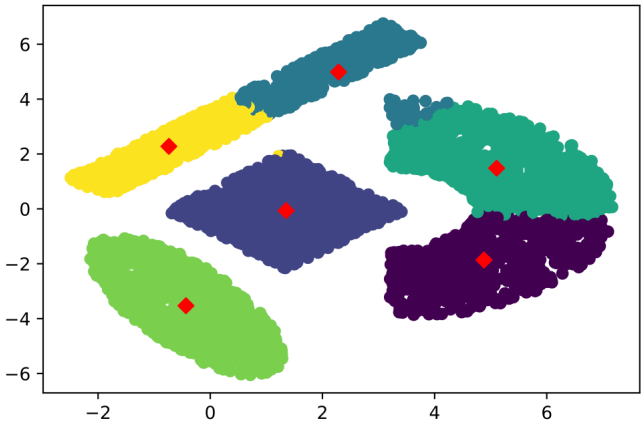
K = 2



K = 4

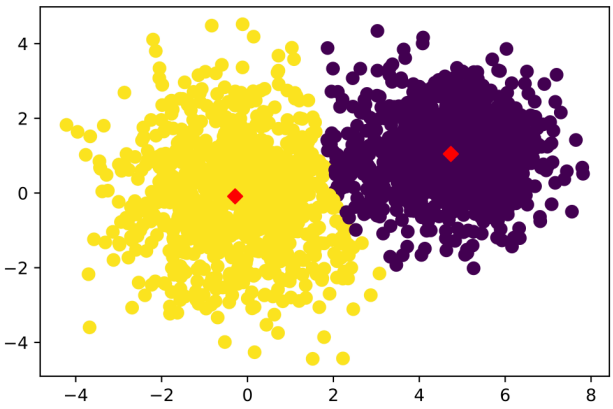


K = 6

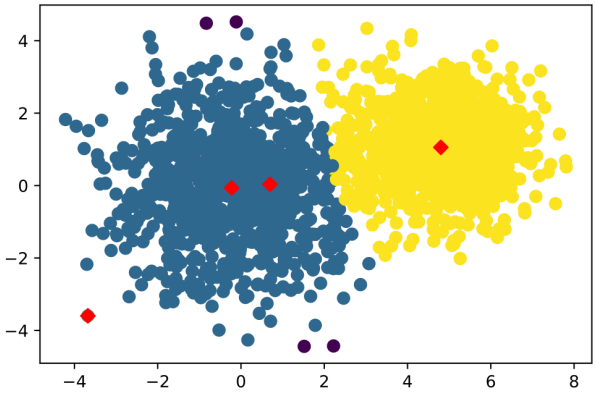


Spectral Clustering
Dataset 1

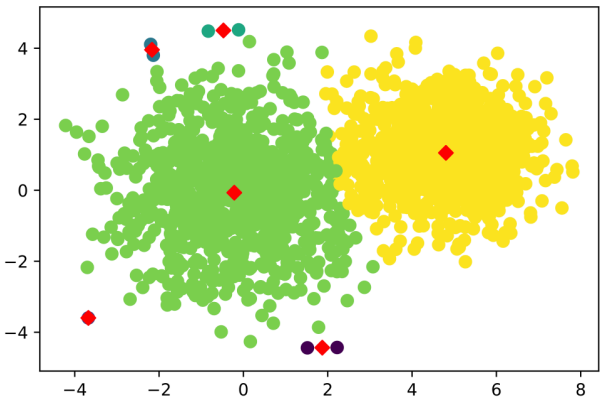
K = 2



K = 4



K = 6

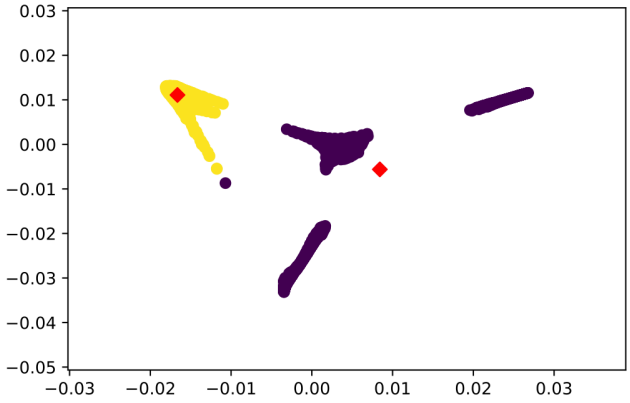


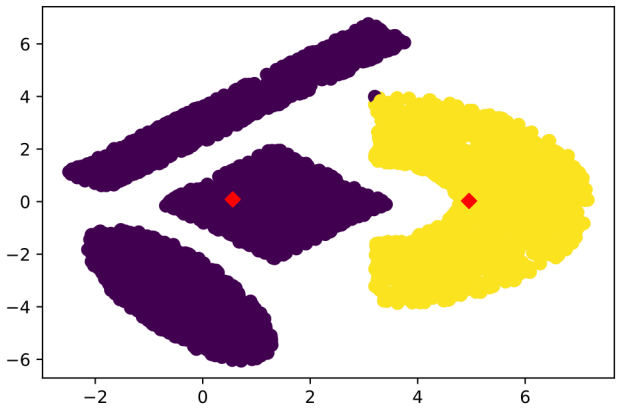
Spectral Clustering

Dataset 2

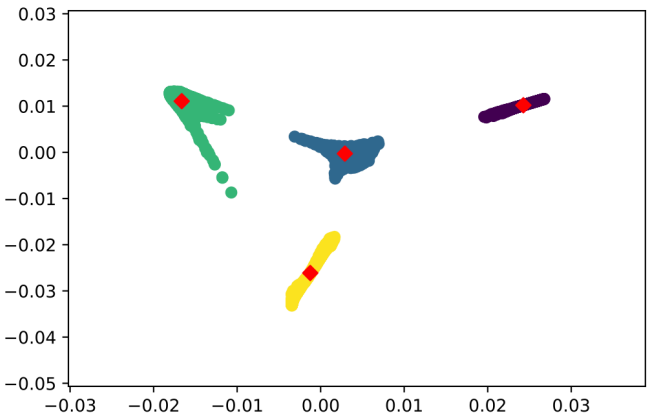
random initializations I implemented for KMeans with Spectral Clustering are 5. Because compute similarity matrix and perform decomposition is very time-consuming and have a high computational complexity.

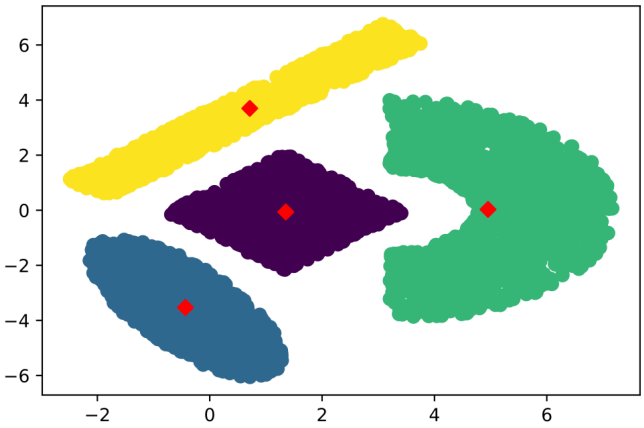
K = 2



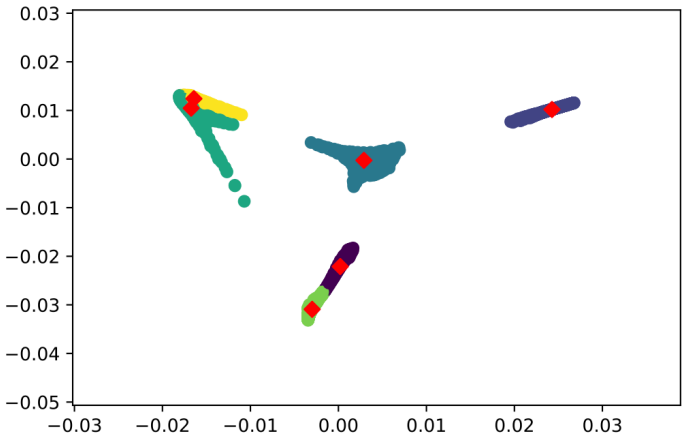


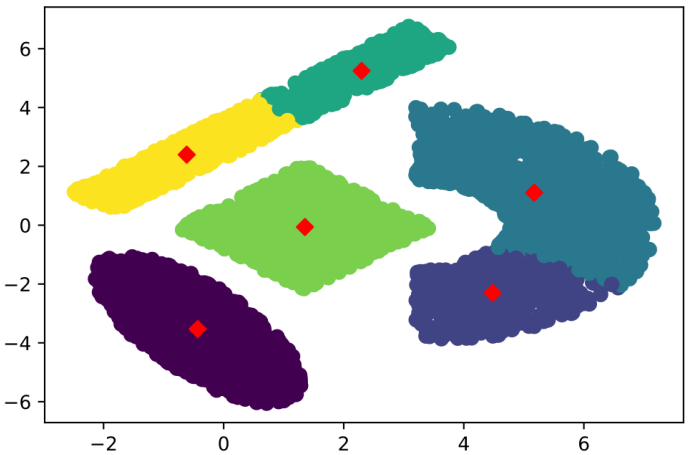
K = 4




6
4
2
0
−2
−4
−6
−2
0
2
4
6

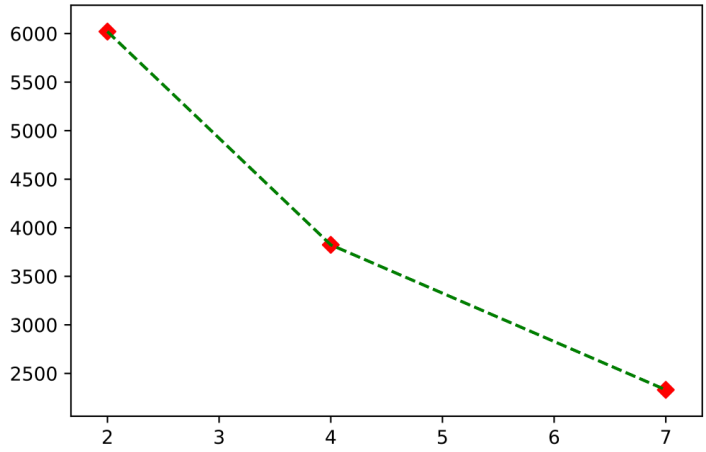
K = 6


0.03
0.02
0.01
0.00
−0.01
−0.02
−0.03
−0.04
−0.05
−0.03
−0.02
−0.01
0.00
0.01
0.02
0.03

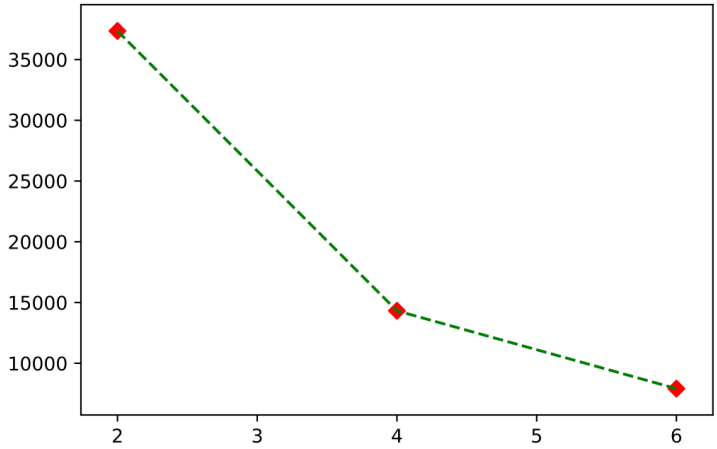

6
4
2
0
−2
−4
−6
−2
0
2
4
6

(2)

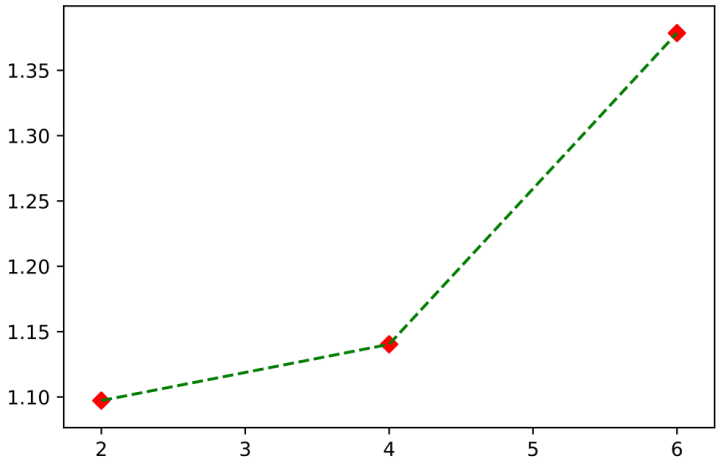
KMeans for Dataset 1



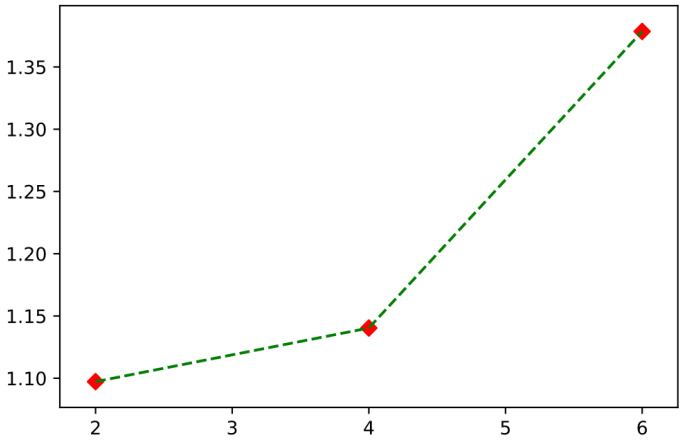
KMeans for Dataset 2



Spectral Clustering for Dataset 1

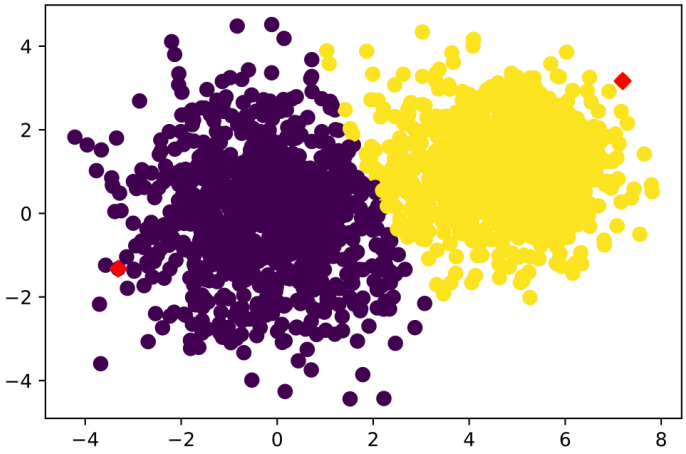


Spectral Clustering for Dataset 2

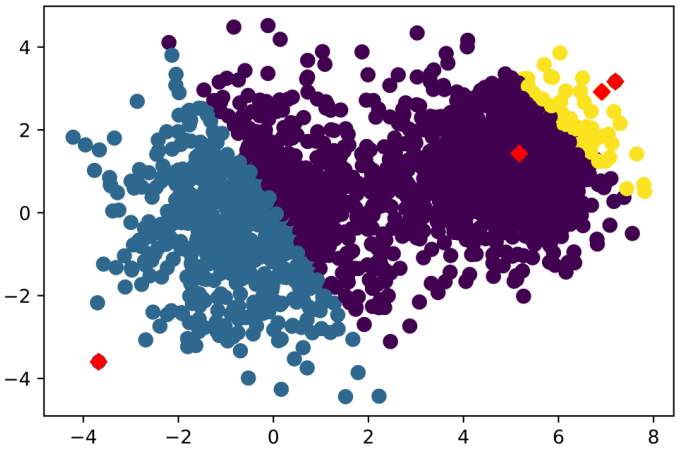


(3)

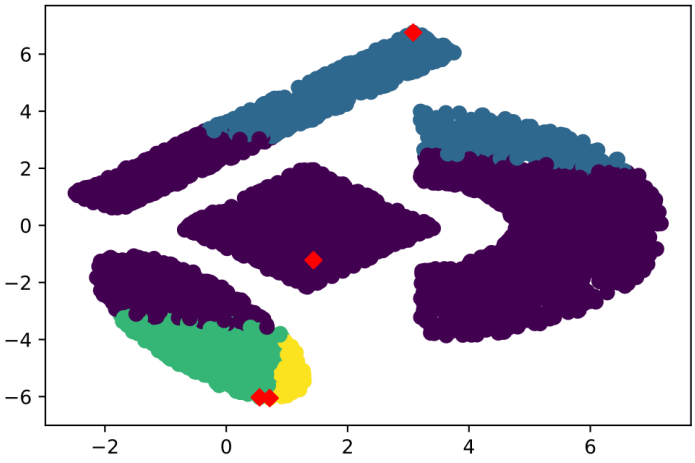
GreedyKCenters, K = 2, Dataset 1



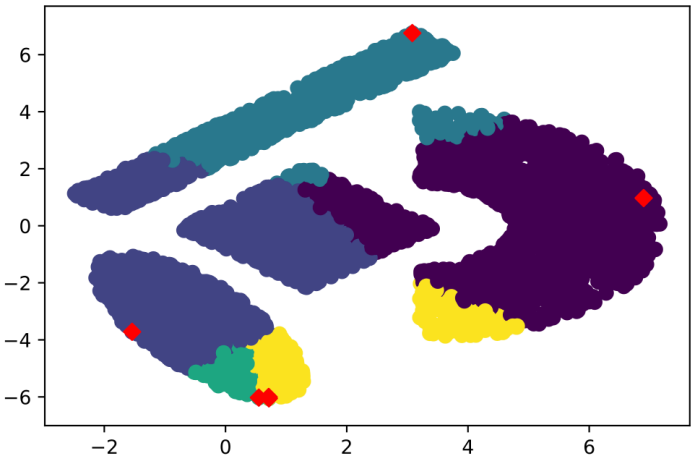
GreedyKCenters, K = 4, Dataset 1



GreedyKCenters, K = 4, Dataset 2



GreedyKCenters, K = 6, Dataset 2



Visually, the first centroid in on the edge of sample data. Those next centroids will be the farthest point from the first one (previous ones). So it does not improve the result with any of two datasets with either K = 2 or K = 4 or K = 6. Anyway, it does not help improve without swapping.

5. For the K-means algorithm, describe how you tested for convergence and why you choose this as a test.

The convergence condition I used is norm-based comparison of the distortion achieved $\| D_{p+1} - D_p \| < tol$. I chose *tol* to be 1×10^{-5} and 0.0001.

The distortion is defined by the the sum of the squared distances between each observation and its closest centroid. So if the distortion in *pth* iteration is very closed to the next iteration, like less than the threshold, we consider the specific number of centroids of this dataset has converged, to some extent. Considering the coordinates of all the data samples, the threshold is much less than the coordinates so if the difference of distortion is less than the threshold. We consider the iterations have converged.

6. Compare the results you get using your spectral clustering algorithm to the results obtained with your K-means algorithms for each of the data sets; you should compare the objective function values (D) and discuss qualitatively how well you think each clustering method worked (i.e., base this comparison on the results given in 4).

	Dataset 1		Dataset 2	
	KMeans	Spectral Clustering	KMeans	Spectral Clustering
K = 2	6020.257	1.112	37350.084	1.097
K = 4	3824.009	1.277	14318.542	1.140

For dataset 1, visually KMeans and Spectral Clustering both give awesome results. Also, we can see the objective function decrease from 6020 to 3824 when K increases from 2 to 4. KMeans works well on 'mixtures of Gaussians' as dataset 1.
For dataset 2, visually speaking, Spectral Clustering gives better result. Because there are four different shapes. KMeans just assigns the nearest centroids to each data point in dataset. Spectral Clustering compute the similarity matrix to evaluate how much similar within the data point.

7. Discuss how many "natural" clusters you think each data set has, and why.

Dataset 1: Two clusters
Dataset 2: Four clusters

Visually, sample data in first dataset is kind of mixed together. So intuitively there are two clusters in dataset 1. For dataset 2, those data points are shaped in specific geometric shapes. So there exist four clusters in dataset 2 obviously.

8. Provide a short analysis of the computational effort you found was required for each of the algorithms. State what measure you used to evaluate computational effort (e.g., running time or flops, or sth.), and how you observed this. Briefly

discuss how these computational efforts compare to your expectations.

Lloyd's Algorithm **O(nkd)** in each iteration: Computational complexity: for each centroid, you should compute the distance between sample data and centroids. The worst case will be $O(n^{(dk+1)}\log n)$.

Spectral Clustering you should compute the similarity matrix and eigenvalue decomposition $\mathbf{O(n^3)}$. When it comes to a large dataset. It is so expensive to compute the similarity matrix and eigenvalue decomposition, like dataset 2 in this assignment.

As the computational complexity of spectral clustering is O(n3). It is obvious when you run the code for spectral clustering, especially the two datasets contain so many observations. The process of computing similarity matrix and decomposition are both time-consuming.

9. Write a brief summary paragraph of your findings, that is, briefly summarize your KEY findings.

Spectral Clustering: Data points as nodes of a connected graph and clusters are found by partitioning this graph, based on its spectral decomposition, into sub-graphs.
K-means clustering: Divide the objects into k clusters such that some metric relative to the centroids of the clusters is minimized.

KMeans handle 'mixtures of Gaussian' perfectly with random initialization of centers. KMeans is ideal for discovering globular clusters like the ones shown below, where all members of each cluster are in close proximity to each other (in the Euclidean sense). Spectral clustering is more general (and powerful) because whenever K-means is appropriate for use then so too is spectral clustering (just use a simple Euclidean distance as the similarity measure). The converse is not true though. With Greedy-K-centers, sometimes it will have stuck at the 'outlier'.

Say if P data points each with N dimensions/features. Then using K-means you'll be dealing with an N by P matrix, while the input matrix to spectral clustering is of size P by P. You should now see the practical implications: spectral clustering is indifferent to the number of features you use (Gaussian kernel which can be thought of as an infinite-dimensional feature transformation is particularly popular when using spectral clustering). However, you will face difficulties applying spectral clustering (at least the vanilla version) to very large datasets (large P).

I. Write a basic implementation of Lloyd's algorithm for a large set of data in R^d (i.e., to find a Voronoi partition and a set of K centroids). Your algorithm should attempt to solve the classic K-means problem, for any user-selected positive integer value K.

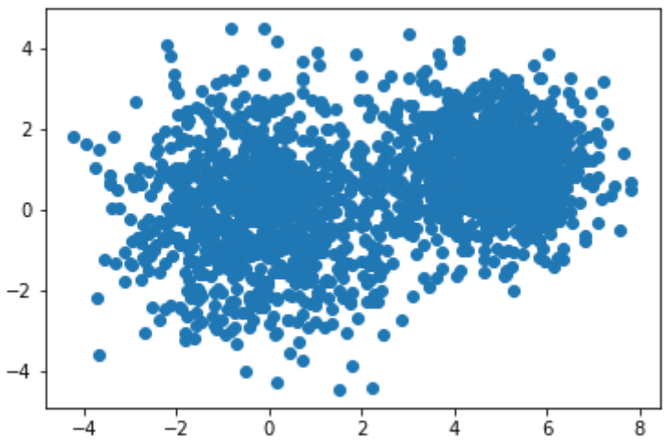
- Assume the input data is given to you in a matrix X ∈ R(N×d), where each row in X corresponds to an observation of a d-dimensional point. That is, your inputs will be a user-provided matrix X and the number of clusters K.
- Your outputs should be (i) a matrix Y ∈ R K×d , where row j contains the centroid of the j th partition; (ii) a cluster index vector C ∈ {1, 2, . . . K} N , where C(i) = j indicates that the i th row of X (or the i th observation x i) belongs to cluster j; and (iii) the final objective function value, i.e., the best distortion, or averaged distance value, D obtained.
- Convergence may be based on a norm-based comparison of the iterates of Y , i.e., kY p+1 –Y p k < tol, OR on a norm-based comparison of the distortion achieved kD p+1 – D p k < tol. Choose tol to be (1) 1 × 10 –5 , and (2) a different value of your choice, with your reasoning provided.

In [32]:
```
import csv
import math
import numpy as np
import matplotlib.pyplot as plt
```

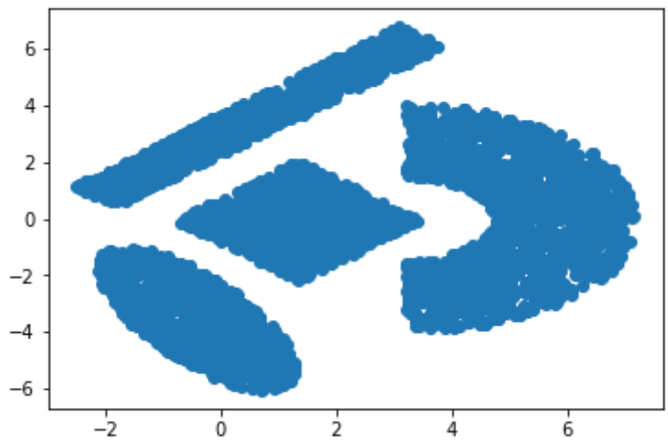
Load dataset and visualize

In [33]:
```
with open('/Users/macbookpro/Desktop/IE529_Comp2/Dataset_1/clustering.csv', 'r') as csvfile:
    reader = csv.reader(csvfile, delimiter=',')
    x = list(reader)
    data_1 = np.array(x).astype("float")

with open('/Users/macbookpro/Desktop/IE529_Comp2/Dataset_2/ShapedData.csv', 'r') as csvfile:
    reader = csv.reader(csvfile, delimiter=',')
    x = list(reader)
    data_2 = np.array(x).astype("float")
```

In [3]:
```
plt.scatter(data_1[:,0],data_1[:,1])
plt.show()
```



In [4]:
```
plt.scatter(data_2[:,0],data_2[:,1])
plt.show()
```



define the function that initialize the centroids

In [50]:
```
# farthest point init centroids

def centroids_init(matrix, K):
    index = [np.random.randint(low=0, high=len(matrix[:,0]))]
    for count in range(1,K):
        for p_index in index:
            x, y = matrix[p_index]
            s = np.array([[0, 0]])
            for num in range(len(matrix[:,0])):
                if num != p_index:
                    distance = math.sqrt((x - matrix[num][0])**2 + (x -
matrix[num][1])**2)
                    t = np.array([[distance, num]])
                    s = np.concatenate((s, t), axis=0)
            s = s[np.argsort(s[:, 0])]
        index.append(int(s[-1][1]))
    centroids = matrix[index]
    return centroids


# # Randomly initialize centroids

# def centroids_init(matrix, K):
#     index = np.random.randint(low=0, high=len(matrix[:,0]), size=K)
#     centroids = matrix[index]
#     return centroids
```

In [51]:
```
def find_closest_centroids(matrix, centroids):
    # Set m
    m = centroids.shape[0]

    # initialize distance matrix
    distance = np.zeros((matrix.shape[0], m))

    for i in range(matrix.shape[0]):
        for j in range(m):
            distance[i][j] = math.sqrt((centroids[j][0] - matrix[i][0])*
*2 + (centroids[j][1] - matrix[i][1])**2)
#             distance[i,:] = np.array([j, math.sqrt((centroids[j,:] - m
atrix[i,:]).dot(centroids[j,:] - matrix[i,:]))])
    idx = np.argmin(distance, axis=1)
#     new_centroids = centroids[idx]
    return idx
```

In [52]:
```
def compute_centroids(matrix, idx, K):
    centroid_x = sum(matrix[np.where(idx == 0)][:,0])/len(matrix[np.wher
e(idx == 0)])
    centroid_y = sum(matrix[np.where(idx == 0)][:,1])/len(matrix[np.wher
e(idx == 0)])
    centroids = np.array([[centroid_x, centroid_y]])
    for i in range(1, K):
        index = matrix[np.where(idx == i)]
        centroid_x, centroid_y = sum(index[:,0])/len(index), sum(index
[:,1])/len(index)
        centroids = np.concatenate((centroids, np.array([[centroid_x, ce
ntroid_y]])))
    return centroids
```

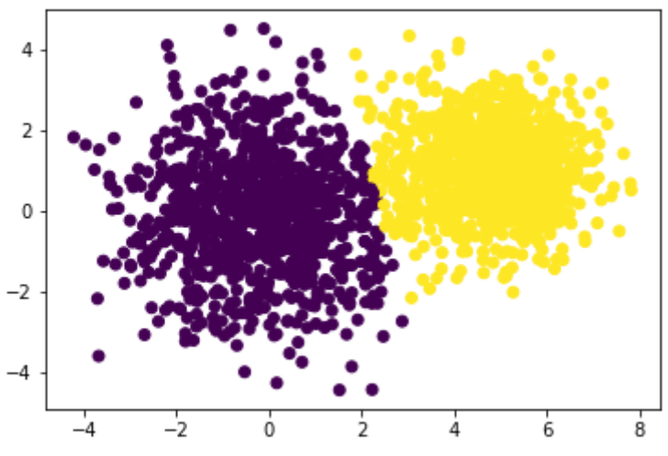
In [53]:
```
def compute_distortion(matrix, idx, centroids, K):
    distance = []
    for i in range(K):
        group = matrix[np.where(idx == i)]
        for j in range(group.shape[0]):
            distance.append(math.sqrt((centroids[i][0] - group[j][0])**2
 + (centroids[i][1] - group[j][1])**2))
    distortion = sum(distance)
    return distortion
```

Experiment on dataset 1

In [54]:
```
# parameters initialization
K = 2
data = data_1
max_iter = 300
val = [0.0001, 10**(-5)]
```

In [55]:
```
# val = 10**-5

centroids = centroids_init(data, K)

for _ in range(max_iter):
    idx = find_closest_centroids(data, centroids)
    new_centroids = compute_centroids(data, idx, K)
    d = compute_distortion(data, idx, centroids, K)
    d_new = compute_distortion(data, idx, new_centroids, K)
    if abs(d_new - d) < val[1]:
        break
    else:
        centroids = new_centroids
        new_centroids = compute_centroids(data, idx, K)
```

In [57]:
```
plt.scatter(data[:,0],data[:,1], c=idx)
plt.show()
```



In [12]:
```
# Centroids for this dataset
Y = centroids
Y
```

Out[12]:
```
array([[ 4.80833513,  1.05385739],
       [-0.2159331 , -0.0629825 ]])
```

In [13]:
```
# Distortion
D = d_new
D
```

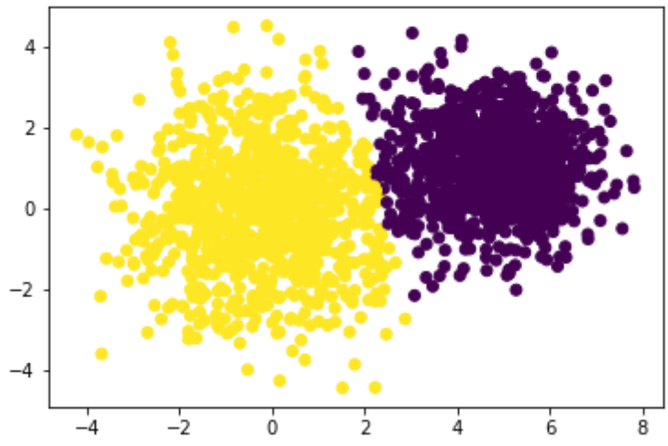
Out[13]:
```
3049.223071649427
```

In [14]:
```
# cluster index vector
C = idx
C
```

Out[14]:
```
array([1, 1, 1, ..., 0, 0, 0])
```

In [15]:
```
# val = 0.0001

centroids = centroids_init(data, K)

for _ in range(max_iter):
    idx = find_closest_centroids(data, centroids)
    new_centroids = compute_centroids(data, idx, K)
    d = compute_distortion(data, idx, centroids, K)
    d_new = compute_distortion(data, idx, new_centroids, K)
    if abs(d_new - d) < val[0]:
        break
    else:
        centroids = new_centroids
        new_centroids = compute_centroids(data, idx, K)
```

In [16]:
```
plt.scatter(data[:,0],data[:,1], c=idx)
plt.show()
```



In [17]:
```
# Centroids for this dataset
Y = centroids
Y
```

Out[17]:
```
array([[ 4.80833513,  1.05385739],
       [-0.2159331 , -0.0629825 ]])
```

In [18]:
```
# Distortion
D = d_new
D
```

Out[18]:
```
3049.223071649427
```

In [19]:
```
# cluster index vector
C = idx
C
```

Out[19]:
```
array([1, 1, 1, ..., 0, 0, 0])
```

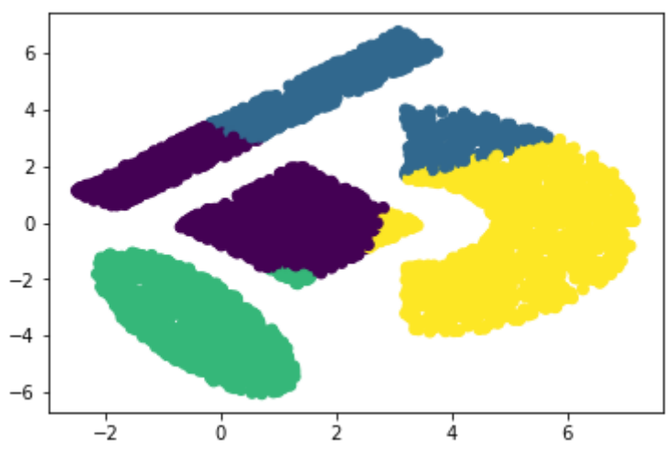
Experiment on dataset 2

In [20]:
```
K = 4
data = data_2
max_iter = 300
val = [0.001, 10**-5]
```

In [21]:
```
# Randomly initialize centroids

def centroids_init(matrix, K):
    index = np.random.randint(low=0, high=len(matrix[:,0]), size=K)
    centroids = matrix[index]
    return centroids



def compute_centroids(matrix, idx, K):
    centroid_x = sum(matrix[np.where(idx == 0)][:,0])/len(matrix[np.wher
e(idx == 0)])
    centroid_y = sum(matrix[np.where(idx == 0)][:,1])/len(matrix[np.wher
e(idx == 0)])
    centroids = np.array([[centroid_x, centroid_y]])
    for i in range(1, K):
        index = matrix[np.where(idx == i)]
#         print (len(index))
        centroid_x, centroid_y = sum(index[:,0])/len(index), sum(index
[:,1])/len(index)
        centroids = np.concatenate((centroids, np.array([[centroid_x, ce
ntroid_y]])))
    return centroids
```

In [22]:
```
# val = 10**-5

centroids = centroids_init(data, K)

for _ in range(max_iter):
    idx = find_closest_centroids(data, centroids)
    new_centroids = compute_centroids(data, idx, K)
    d = compute_distortion(data, idx, centroids, K)
    d_new = compute_distortion(data, idx, new_centroids, K)
    if abs(d_new - d) < val[1]:
        break
    else:
        centroids = new_centroids
        new_centroids = compute_centroids(data, idx, K)
```

In [23]:
```
plt.scatter(data[:,0],data[:,1], c=idx)
plt.show()
```



In [24]:
```
# Centroids for this dataset
Y = centroids
Y
```

Out[24]:
```
array([[ 0.55342828,  0.62239439],
       [ 2.50822746,  4.29537602],
       [-0.38157179, -3.48083101],
       [ 5.02221541, -0.49046381]])
```

In [25]:
```
# Distortion
D = d_new
D
```

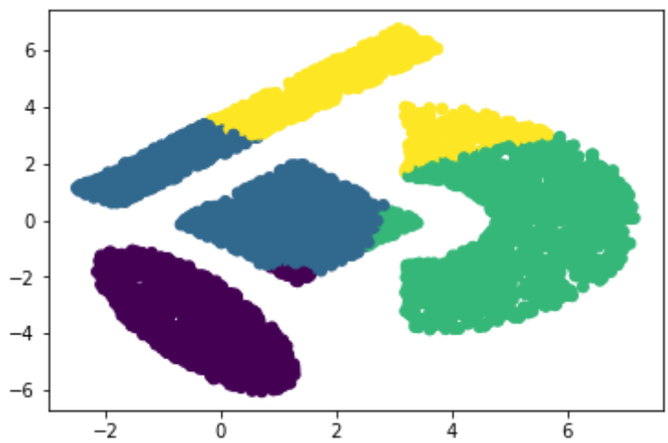
Out[25]:
```
7337.5390037174175
```

In [26]:
```
# cluster index vector
C = idx
C
```

Out[26]:
```
array([0, 3, 3, ..., 1, 3, 2])
```

In [27]:
```
# val = 10**-5

centroids = centroids_init(data, K)

for _ in range(max_iter):
    idx = find_closest_centroids(data, centroids)
    new_centroids = compute_centroids(data, idx, K)
    d = compute_distortion(data, idx, centroids, K)
    d_new = compute_distortion(data, idx, new_centroids, K)
    if abs(d_new - d) < val[0]:
        break
    else:
        centroids = new_centroids
        new_centroids = compute_centroids(data, idx, K)
```

In [28]:
```
plt.scatter(data[:,0],data[:,1], c=idx)
plt.show()
```



In [29]:
```
# Centroids for this dataset
Y = centroids
```

In [30]:
```
# Distortion
D = d_new
```

In [31]:
```
# cluster index vector
C = idx
C
```

Out[31]:
```
array([1, 2, 2, ..., 2, 2, 0])
```

In [ ]:

In [ ]:

II. (1) Write a basic implementation of the ”GreedyKCenters” algorithm (described in the reading by S. Har-Peled, and discussed in class). Your algorithm should attempt to solve the classic K-centers problem, for any user-selected positive integer value K. The underlying distance function used in your algorithm should be the Euclidean distance, and your objective should be to minimize the maximum distance between any observation x i ∈ X and it’s closest center c j ∈ Q, i.e., to find Q giving

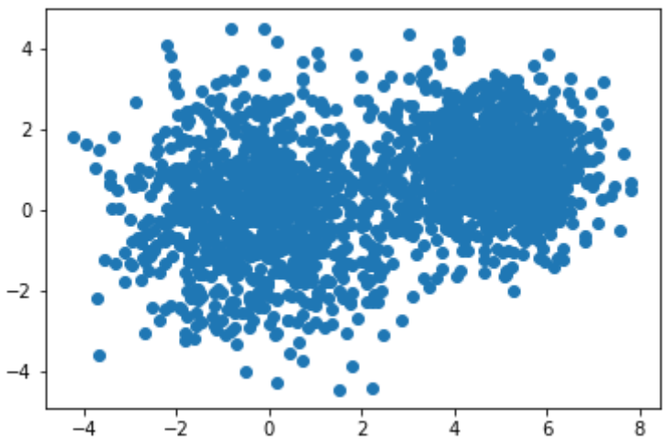
- You can again assume the input data is given to you as a matrix X ∈ R N×d , and a positive integer K, as in I.
- Your output should be a matrix Q ∈ R K×d containing the final K d-dimensional centers, and the objective function value, i.e., the final max x i ∈X (min c j ∈Q kx i – c j k 2) obtained.

In [40]:
```
import csv
import math
import numpy as np
from numpy.linalg import norm
import matplotlib.pyplot as plt
```

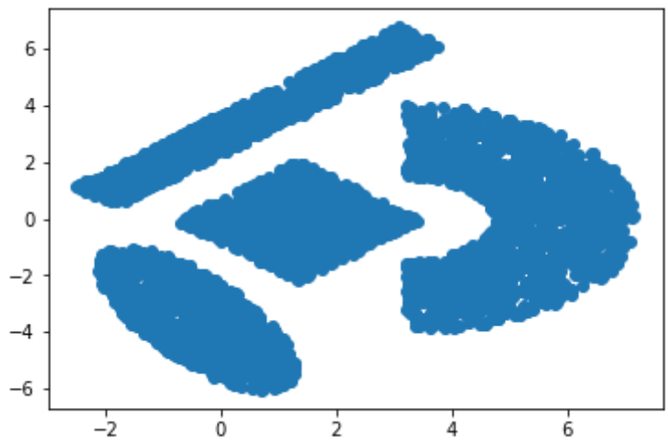
Data visualization

In [41]:
```
with open('/Users/macbookpro/Desktop/IE529_Comp2/Dataset_1/clustering.csv', 'r') as csvfile:
    reader = csv.reader(csvfile, delimiter=',')
    x = list(reader)
    data_1 = np.array(x).astype("float")


with open('/Users/macbookpro/Desktop/IE529_Comp2/Dataset_2/ShapedData.csv', 'r') as csvfile:
    reader = csv.reader(csvfile, delimiter=',')
    x = list(reader)
    data_2 = np.array(x).astype("float")
```

In [42]:
```
plt.scatter(data_1[:,0],data_1[:,1])
plt.show()
```



In [43]:
```
plt.scatter(data_2[:,0],data_2[:,1])
plt.show()
```



Define functions

In [44]:
```
# def centroids_init(matrix, K):
#     index = [np.random.randint(low=0, high=len(matrix[:,0]))]
#     for count in range(1,K):
#         for p_index in index:
#             x, y = matrix[p_index]
#             s = np.array([[0, 0]])
#             for num in range(len(matrix[:,0])):
#                 if num != p_index:
#                     distance = math.sqrt((x - matrix[num][0])**2 + (x
 - matrix[num][1])**2)
#                     t = np.array([[distance, num]])
#                     s = np.concatenate((s, t), axis=0)
#             s = s[np.argsort(s[:, 0])]
#         index.append(int(s[-1][1]))
#     centroids = matrix[index]
#     return centroids


# Randomly initialize centroids
def centroids_init(matrix, K):
    index = np.random.randint(low=0, high=len(matrix[:,0]), size=K)
    centroids = matrix[index]
    return centroids
```

In [45]:
```
def find_farthest_point(X, Q):
    maxDist = 0
    maxIndex = 0
    for j in range(Q.shape[0]):
        for i in range(X.shape[0]):
            d = math.pow(norm((X[i]-Q[j]),2), 2)
            if (d >= maxDist):
                maxDist = d
                maxIndex = i
    return (maxIndex, X[[maxIndex]], maxDist)
```

In [46]:
```
def GreedyKCenters(X, Q):
    objFuncs = []
    for i in range(1, Q.shape[0]):
        Index, NewCentroids, Dist = find_farthest_point(X, Q)
        # Assign nee centroids
        Q[i] = NewCentroids
        # Remove the point which assigned to centroids
        X = np.delete(X, Index, axis = 0)
        # objective function -> L2 norm distance
        objFuncs.append(Dist)
    return Q, objFuncs[-1]
```

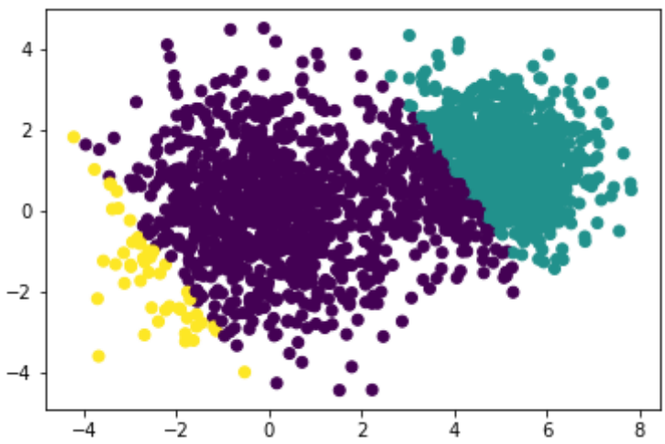
In [47]:
```
def find_closest_centroids(matrix, centroids):
    # Set m
    m = centroids.shape[0]

    # initialize distance matrix
    distance = np.zeros((matrix.shape[0], m))

    for i in range(matrix.shape[0]):
        for j in range(m):
            distance[i][j] = math.sqrt((centroids[j][0] - matrix[i][0])*
*2 + (centroids[j][1] - matrix[i][1])**2)
    idx = np.argmin(distance, axis=1)
#     new_centroids = centroids[idx]
    return idx
```

Play with the first dataset

In [48]:
```
# Parameters settings
data = data_1
K = 3
firstCentroids = centroids_init(data, 1)
Q = np.zeros((K, 2))
Q[0] = firstCentroids
```

In [49]:
```
centroids, dist = GreedyKCenters(data, Q)
idx = find_closest_centroids(data, centroids)
```

In [50]:
```
plt.scatter(data[:,0],data[:,1], c=idx)
plt.show()
```



In [51]:
```
# Centroids
centroids
```

Out[51]:
```
array([[ 0.57365, -0.94986],
       [ 7.1976 ,  3.1657 ],
       [-3.6773 , -3.5965 ]])
```

In [52]:
```
# objective function
dist
```

Out[52]:
```
163.99079884999998
```

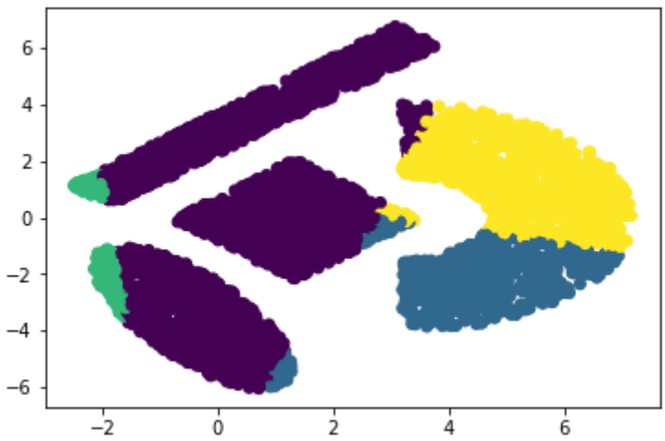
In [ ]:
```

```

Play with the second dataset

In [53]:
```
# Parameters settings
data = data_2
K = 4
firstCentroids = centroids_init(data, 1)
Q = np.zeros((K, 2))
Q[0] = firstCentroids
```

In [54]:
```
centroids, dist = GreedyKCenters(data, Q)
idx = find_closest_centroids(data, centroids)
```

In [55]:
```
plt.scatter(data[:,0],data[:,1], c=idx)
plt.show()
```



In [56]:
```
# Centroids
centroids
```

Out[56]:
```
array([[-1.443  ,  1.2215 ],
       [ 6.8767 , -1.5744 ],
       [-2.4871 ,  1.1324 ],
       [ 7.0904 , -0.82791]])
```

In [57]:
```
# objective function
dist
```

Out[57]:
```
95.57132154610002
```

In [ ]:

In [ ]:

In [ ]:

II. (2) Write a basic implementation of the single-swap heuristic for which you try to improve the solution to the K-centers problem in II.1 by a implementing a series of ”swaps”. If Q is your current set of centers, and you make a single swap, giving Q new = Q – {c j } ∪ {o}, then you should replace Q with Q whenever the new objective value, that is the computed value for (1), is reduced by a factor of new (1 – τ). When there is no swap that improves the solution by this factor, the local search stops. Let τ = 0.05.

In [58]:
```
def swap(X, Q):
    dummy1, dummy2, cost = find_farthest_point(X, Q)
    cond = True
    while cond:
        new_Q = np.zeros(Q.shape)
        if Q.shape[0] == 0:
            cond = False
        i = np.random.randint(0,X.shape[0])
        j = np.random.randint(0,Q.shape[0]) # Centroids
        Q = np.delete(Q, j, axis=0)
        Q = np.append(Q, [X[i]], axis=0)
        X = np.delete(X, i, axis=0)
        dummy1, dummy2, new_cost = find_farthest_point(X, Q)
        if new_cost < cost:
            cond = False
            Flag = False
            break
    return X, Q, new_cost, Flag
```

Play with the second dataset

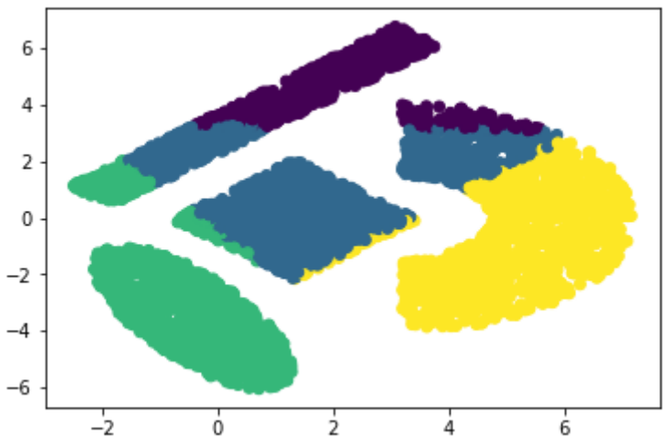
In [24]:
```
centroids, dist = GreedyKCenters(data, Q)
print (centroids)
print ("")
print (dist)
```

```
[[ 2.0709   5.3964 ]
 [ 3.0811   6.7565 ]
 [ 0.54923 -6.0252 ]
 [ 0.71578 -6.0515 ]]

169.63960270240003
```

In [ ]:
```
```

In [31]:
```
cond = True
count = 0
t = 0.05
objFunc = []
while cond:
    new_X, new_Q, cost, flag = swap(data, centroids)
    if (flag == True):
        cond = False
    objFunc.append(cost)
    if (count > 0):
        if (objFunc[count] <= (1- t) * objFunc[count-1]):
            cond = False
    count += 1
```

In [36]:
```
idx = find_closest_centroids(new_X, new_Q)
plt.scatter(new_X[:,0],new_X[:,1], c=idx-1)
plt.show()
```



In [37]:
```
new_Q
```

Out[37]:
```
array([[ 2.0709,  5.3964],
       [ 1.9128,  1.0191],
       [-2.0769, -1.6016],
       [ 4.4443, -1.375 ]])
```

In [38]:
```
objFunc
```

Out[38]:
```
[169.63960270240003, 165.13471652689992, 132.89076462440002]
```

In [ ]:

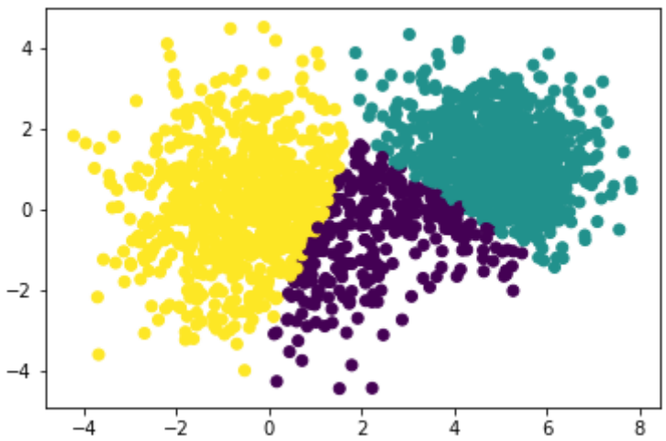
In [ ]:

Play with the first dataset

In [59]:
```
# Parameters settings
data = data_1
K = 3
firstCentroids = centroids_init(data, 1)
Q = np.zeros((K, 2))
Q[0] = firstCentroids
centroids, dist = GreedyKCenters(data, Q)
```

In [60]:
```
cond = True
count = 0
t = 0.05
objFunc = []
while cond:
    new_X, new_Q, cost, flag = swap(data, centroids)
    if (flag == True):
        cond = False
    objFunc.append(cost)
    if (count > 0):
        if (objFunc[count] <= (1- t) * objFunc[count-1]):
            cond = False
    count += 1
```

In [61]:
```
idx = find_closest_centroids(new_X, new_Q)
plt.scatter(new_X[:,0],new_X[:,1], c=idx-1)
plt.show()
```



In [62]:
```
new_Q
```

Out[62]:
```
array([[ 2.3282 , -0.93878],
       [ 4.6583 ,  2.0664 ],
       [-0.21275,  0.1002 ]])
```

In [63]:
```
objFunc
```

Out[63]:
```
[116.92489492999998, 101.55066376999999]
```

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

III. Write an implementation of the Spectral Clustering algorithm, using either basic unnormalized clustering or normalized clustering (refer to the reading by Luxborg for details). Assume you are given a matrix of data X ∈ R N×d , and you would like to identify some user-selected number of clusters, K. Your outputs should be:

- a weighted adjacency matrix, W, using the Gaussian similarity function based on the Eu- clidean distance (with parameter value σ of your choice but clearly stated) and a k-nearest neighborhood structure (where k is also your choice and clearly stated);
- a matrix U containing the first K eigenvectors of the Laplacian L (or generalized eigenvectors for the normalized case).;
- a cluster index vector C ∈ {1, 2, . . . , K} N , where C(i) = j indicates that the ith row of U belongs to cluster j.

basic unnormalized clustering

In [106]:
```
import csv
import math
import numpy as np
import scipy as sp
from scipy.linalg import eigh
from numpy.linalg import norm
import matplotlib.pyplot as plt
from sklearn.neighbors import NearestNeighbors
```

In [107]:
```
with open('/Users/macbookpro/Desktop/IE529_Comp2/Dataset_1/clustering.csv', 'r') as csvfile:
    reader = csv.reader(csvfile, delimiter=',')
    x = list(reader)
    data_1 = np.array(x).astype("float")


with open('/Users/macbookpro/Desktop/IE529_Comp2/Dataset_2/ShapedData.csv', 'r') as csvfile:
    reader = csv.reader(csvfile, delimiter=',')
    x = list(reader)
    data_2 = np.array(x).astype("float")
```

Define adj matrix

In [109]:
```
# Gaussian similarity function
# For example, one can choose σ in the order of the mean distance of a point to its k-th nearest neighbor,
# where k is chosen similarly as above (e.g., k ~ log(n) + 1 ). like σ=5
def adj_generate(X, gamma):
    n = X.shape[0]
    a = np.zeros([n,n])
    for i in range(n):
        for j in range(n):
            a[i,j] = math.exp(-math.pow(norm((X[i]-X[j]),2), 2) * gamma)
    return a
```

In [108]:
```
# k-nearest neighborhood structure
# k -> log(n) -> 3 or 4
def adj_generate_KNN(X, k):
    n = X.shape[0]
    a = np.zeros([n,n])
    neigh = NearestNeighbors(n_neighbors=k)
    neigh.fit(X)
    idx = neigh.kneighbors(X,  n_neighbors=k, return_distance=False)
    for i in range(n):
        for j in range(n):
            if j in idx[i]:
                a[i,j] = 1
    return a
```

In [110]:
```
def diag_generate(a):
    n = a.shape[0]
    d = np.sum(a, axis = 1)
    return np.diagflat(d)
```

Kmeans below

In [111]:
```
# Randomly initialize centroids
def centroids_init(matrix, K):
    index = np.random.randint(low=0, high=len(matrix[:,0]), size=K)
    centroids = matrix[index]
    return centroids
```

In [135]:
```
def find_closest_centroids(matrix, centroids):
    # Set m
    m = centroids.shape[0]

    # initialize distance matrix
    distance = np.zeros((matrix.shape[0], m))

    for i in range(matrix.shape[0]):
        for j in range(m):
#             distance[i][j] = math.sqrt((centroids[j][0] - matrix[i]
[0])**2 + (centroids[j][1] - matrix[i][1])**2)
            distance[i,j] = math.pow(norm((matrix[i,:]-centroids[j,:]),2
), 0.5)
    idx = np.argmin(distance, axis=1)
#     new_centroids = centroids[idx]
    return idx
```

In [191]:
```
def compute_centroids(matrix, idx, K):
    temp = []
    for j in range(matrix.shape[1]):
        a = matrix[np.where(idx == 0)][:,j].sum()
        b = len(matrix[np.where(idx == 0)])
        temp.append(a/b)
    centroids = np.array([temp])
    for i in range(1, K):
        index = matrix[np.where(idx == i)]
        temp = []
        for i in range(matrix.shape[1]):
            temp.append(index[:,i].sum()/len(index))
        centroids = np.concatenate((centroids, np.array([temp])))
    return centroids
```

In [201]:
```
def compute_distortion(matrix, idx, centroids, K):
    distance = []
    for i in range(K):
        group = matrix[np.where(idx == i)]
        for j in range(group.shape[0]):
            distance.append(math.pow(norm((centroids[i,:]-group[j,:]),2
), 2))
    distortion = 0
    for i in distance:
        distortion += i
    return distortion
```

Play with Dataset 2 with Gaussian similarity function

Kmeans on spectral clustering

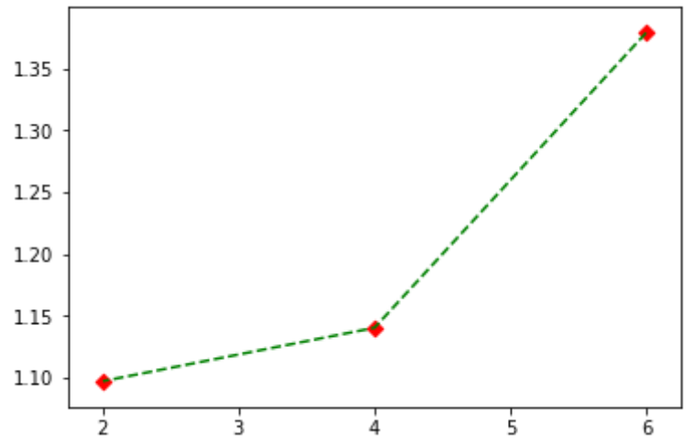
In [348]:
```
X = data_2
K = 6
gamma = 1
```

In [349]:
```
# Adjacency matrix
A = adj_generate(X, gamma)
# degree-diagonal matrix
D = diag_generate(A)


# Unnormalized clustering, Lapalacian matrix
L = D - A

# # Normalized Lapalacian matrix
# def normalize_adj(A, D):
#     d_inv_sqrt = np.power(D, -0.5)
#     d_inv_sqrt[np.isinf(d_inv_sqrt)] = 0.
#     L = sp.eye(D.shape[0]) - d_inv_sqrt.dot(A.dot(d_inv_sqrt))
#     return L
# L = normalize_adj(A, D)

eigValue, U = np.asarray(eigh(L, eigvals=(1,K)))
```

In [327]:
```
# parameters initialization
max_iter = 300
val = [0.001, 10**(-5)] # val = 10**-5

centroids = centroids_init(U, K)

for _ in range(max_iter):
    idx = find_closest_centroids(U, centroids)
    new_centroids = compute_centroids(U, idx, K)
    d = compute_distortion(U, idx, centroids, K)
    d_new = compute_distortion(U, idx, new_centroids, K)
    if abs(d_new - d) < val[1]:
        break
    else:
        centroids = new_centroids
        new_centroids = compute_centroids(U, idx, K)
```

In [ ]:
```

```

In [358]:
```
plt.plot(K_list, inertia, 'g--')
K_list = K_list[::-1]
inertia = inertia[::-1]
plt.scatter(K_list, inertia, c='r', marker = 'D')
plt.savefig('D_K.svg',format='svg')
plt.show()
```



In [ ]:
```
```

In [234]:
```
U
```

Out[234]:
```
array([[ 0.00196131,  0.00126425, -0.02279465, -0.01080707],
       [-0.01753502,  0.01297361,  0.00511127,  0.021116  ],
       [-0.01696202,  0.01281366,  0.003664  ,  0.02615788],
       ...,
       [-0.01552786,  0.00891799,  0.00610137, -0.02019692],
       [-0.01651182,  0.01054058,  0.00645384, -0.01175206],
       [ 0.02509135,  0.01065055,  0.01192572,  0.00288991]])
```

In [235]:
```
A
```

Out[235]:
```
array([[  1.00000000e+00,   2.17643668e-07,   4.55907711e-05, ...,
          2.00765232e-06,   1.21849549e-05,   1.20500035e-07],
       [  2.17643668e-07,   1.00000000e+00,   2.95233116e-01, ...,
          1.51993327e-10,   1.21553359e-06,   1.27463324e-20],
       [  4.55907711e-05,   2.95233116e-01,   1.00000000e+00, ...,
          5.54987749e-11,   2.33137852e-07,   9.67884363e-15],
       ...,
       [  2.00765232e-06,   1.51993327e-10,   5.54987749e-11, ...,
          1.00000000e+00,   2.93388113e-01,   2.00118945e-25],
       [  1.21849549e-05,   1.21553359e-06,   2.33137852e-07, ...,
          2.93388113e-01,   1.00000000e+00,   3.24640735e-24],
       [  1.20500035e-07,   1.27463324e-20,   9.67884363e-15, ...,
          2.00118945e-25,   3.24640735e-24,   1.00000000e+00]])
```

In [ ]:
```
```

In [ ]:
```
```

Play with Dataset 1 with Gaussian similarity function

Kmeans on spectral clustering

In [360]:
```
X = data_1
K = 2
gamma = 1
```

In [361]:
```
# Adjacency matrix
A = adj_generate(X, gamma)
# degree-diagonal matrix
D = diag_generate(A)


# Unnormalized clustering, Lapalacian matrix
L = D - A

# # Normalized Lapalacian matrix
# def normalize_adj(A, D):
#     d_inv_sqrt = np.power(D, -0.5)
#     d_inv_sqrt[np.isinf(d_inv_sqrt)] = 0.
#     L = sp.eye(D.shape[0]) - d_inv_sqrt.dot(A.dot(d_inv_sqrt))
#     return L
# L = normalize_adj(A, D)

eigValue, U = np.asarray(eigh(L, eigvals=(1,K)))
```

In [224]:
```
# parameters initialization
max_iter = 300
val = [0.001, 10**(-5)] # val = 10**-5

centroids = centroids_init(U, K)

for _ in range(max_iter):
    idx = find_closest_centroids(U, centroids)
    new_centroids = compute_centroids(U, idx, K)
    d = compute_distortion(U, idx, centroids, K)
    d_new = compute_distortion(U, idx, new_centroids, K)
    if abs(d_new - d) < val[1]:
        break
    else:
        centroids = new_centroids
        new_centroids = compute_centroids(U, idx, K)
```

In [ ]:
```
C = idx
C
```

In [241]:
```
U
```

Out[241]:
```
array([[-0.00015688, -0.00401449],
       [-0.00020957, -0.00404215],
       [-0.0002489 , -0.00415792],
       ...,
       [ 0.00129351,  0.00572052],
       [ 0.00121567,  0.00514439],
       [ 0.00131331,  0.00588473]])
```

In [242]:
```
A
```

Out[242]:
```
array([[  1.00000000e+00,   2.48160734e-01,   3.06462066e-02, ...,
          9.98950309e-25,   8.52336992e-10,   2.33620199e-18],
       [  2.48160734e-01,   1.00000000e+00,   6.23085894e-01, ...,
          1.54962426e-30,   6.53013575e-13,   2.40788949e-19],
       [  3.06462066e-02,   6.23085894e-01,   1.00000000e+00, ...,
          9.86791506e-35,   1.81971310e-15,   8.71677605e-21],
       ...,
       [  9.98950309e-25,   1.54962426e-30,   9.86791506e-35, ...,
          1.00000000e+00,   8.47445696e-05,   1.04634937e-10],
       [  8.52336992e-10,   6.53013575e-13,   1.81971310e-15, ...,
          8.47445696e-05,   1.00000000e+00,   2.56572453e-05],
       [  2.33620199e-18,   2.40788949e-19,   8.71677605e-21, ...,
          1.04634937e-10,   2.56572453e-05,   1.00000000e+00]])
```

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]: